Learning goals

Before the next day, you should have achieved the following learning goals:

- Use the main classes from package `java.util.concurrent`.
- Understand the concepts of *thread pool* and *graceful degradation*.

1 Graceful degradation everywhere

Think of three examples of concurrent applications that you use every week and that benefit from graceful degradation, i.e. becoming gradually more slow under high load rather than crashing unexpectedly.

Once you have thought of three examples, check them with one of your colleagues. Have you thought of the same examples?

2 Text loops re-executed

Modify the code of the exercise "Text Loops" from last day to use one of the `Executor` instead of plain threads.

3 Responsive UI (that degrades gracefully)

3.1 Implement Executor

Create your own implementation of `Executor`. Look at the documentation of the API of `Executor` for ideas. Your implementation must be able to execute one `Runnable` at a time (but you can make it run several tasks in paralel, see also Exercise 6) and must have a queue of pending tasks.

Try your implementation by exchaning the class that you used in your solution for the former exercise by your new implementation of `Executor`.

3.2 Extend Executor

Create a class `TimedTask` that implements `Runnable` where the `run()` method is only a call to `Thread.sleep()` for a number of milliseconds set at creation time. This number must be at most 1000 (if a higher number is given at construction time, it must be capped at 1000).

Now extend your implementation of `Executor` so that it has a public method `getMaxPendingTime()` that returns the maximum theoretical time needed to run all tasks in the queue (i.e assuming all of them last 1000ms).

3.3 Use Executor

Modify your solution of exercise "Responsive UI" so that:

- It has two classes: one represents the application and one represents its users.
- The former uses your implementation of `Executor` with a pool of threads instead of using plain threads.
- The latter will use an `Executor` from the Java Library[1] to have a lot of threads representing users, and the threads will programatically create new tasks instead of the human user doing it by hand, i.e. there is no need to ask the user to enter data at any point in this version of the exercise. Threads should create tasks as fast as the time they need to be run (e.g. if a task that will make the thread sleep for one second, the next task should be created a second later).
- The "application" class should implement a method `getMaxWait()` that returns the (theoretical) time needed to execute all tasks already in the queue.

[1] Any of the provided implementations will be fine. Choose any of them and guess appropriate values for the initialisation parameters. Ask the faculty members if you need help with this.

- The "users" (threads) will ask for the waiting time before they assign a new task. If the wait is above a certain threshold (set at construction time for each "user"), the "user" should print "The site is down! I will come back later..." and then wait for a long time before sending more tasks.

Try your implementation with different numbers of users and see how many users it can handle.

4 Self-ordering list (*)

Re-implement your solutions for the exercise "Self-ordering list" by using an `Executor`.

5 Dining philosophers (*)

Re-implement your solutions for the exercise "Self-ordering list" by using an `Executor`.

6 Implementing Executor Services (**)

Create your own implementation of `Executor`, `ExecutorService` and `ScheduledExecutorService`, with a thread pool of n, where n is set at construction time. Use your implementation in the former exercises.